Package ‘bbotk’

December 8, 2022

Title Black-Box Optimization Toolkit

Version 0.7.2

Description Features highly configurable search spaces via the ‘paradox’ package and optimizes every user-defined objective function. The package includes several optimization algorithms e.g. Random Search, Iterated Racing, Bayesian Optimization (in ‘mlr3mbo’) and Hyperband (in ‘mlr3hyperband’). bbotk is the base package of ‘mlr3tuning’, ‘mlr3select’ and ‘miesmuschel’.

License LGPL-3


BugReports https://github.com/mlr-org/bbotk/issues

Depends paradox (>= 0.7.0), R (>= 3.1.0)

Imports checkmate (>= 2.0.0), data.table, lgr, methods, mlr3misc (>= 0.11.0), R6

Suggests adagio, emoa, GenSA, irace (>= 3.5), knitr, nloptr,
progressr, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/testthat/edition 3

Config/testthat/parallel false

Encoding UTF-8

Language en-US

NeedsCompilation yes

RoxygenNote 7.2.2

Collate 'Archive.R' 'ArchiveBest.R' 'CallbackOptimization.R'
'Codomain.R' 'ContextOptimization.R' 'Objective.R'
'ObjectiveRFun.R' 'ObjectiveRFunDt.R' 'ObjectiveRFunMany.R'
'OptimInstance.R' 'OptimInstanceMultiCrit.R'
'OptimInstanceSingleCrit.R' 'mlr_optimizers.R' 'Optimizer.R'
'OptimizerCmaes.R' 'OptimizerDesignPoints.R'
'OptimizerFocusSearch.R' 'OptimizerGenSA.R'
R topics documented:

'OptimizerGridSearch.R' 'OptimizerIrace.R' 'OptimizerNLopt.R'
'OptimizerRandomSearch.R' 'Progressor.R' 'mlr_terminators.R'
'TerminatorEvals.R' 'TerminatorNone.R'
'TerminatorPerfReached.R' 'TerminatorRunTime.R'
'assertions.R' 'bb_optimize.R' 'bbotk_reflections.R'
'bibentries.R' 'helper.R' 'mlr_callbacks.R' 'nds_selection.R'
'reexport.R' 'sugar.R' 'zzz.R'

Author  Marc Becker [cre, aut] (<https://orcid.org/0000-0002-8115-0400>),
        Jakob Richter [aut] (<https://orcid.org/0000-0003-4481-5554>),
        Michel Lang [aut] (<https://orcid.org/0000-0001-9754-0393>),
        Bernd Bischl [aut],
        Martin Binder [aut],
        Olaf Mersmann [ctb]

Maintainer Marc Becker <marcbecker@posteo.de>

Repository CRAN

Date/Publication  2022-12-08 08:20:02 UTC

R topics documented:

bbotk-package .................................................. 3
Archive .......................................................... 4
ArchiveBest ......................................................... 7
bbotk.backup ...................................................... 8
bb_optimize ......................................................... 9
branin .............................................................. 11
CallbackOptimization ............................................. 12
callback_optimization ............................................ 13
Codomain ........................................................... 14
ContextOptimization .............................................. 16
is_dominated ....................................................... 17
mlr_optimizers ..................................................... 18
mlr_optimizers_cmaes .......................................... 18
mlr_optimizers_design_points .................................. 20
mlr_optimizers_focus_search .................................. 22
mlr_optimizers_gensa ........................................... 24
mlr_optimizers_grid_search .................................... 26
mlr_optimizers_irace ............................................ 28
mlr_optimizers_nlopt ............................................ 31
mlr_optimizers_random_search .................................. 33
mlr_terminators ................................................. 34
mlr_terminators_clock_time .................................... 35
mlr_terminators_combo .......................................... 37
mlr_terminators_evals .......................................... 39
mlr_terminators_none ........................................... 40
bbotk-package

bbotk: Black-Box Optimization Toolkit

Description

Features highly configurable search spaces via the paradox package and optimizes every user-defined objective function. The package includes several optimization algorithms e.g. Random Search, Iterated Racing, Bayesian Optimization (in mlr3mbo) and Hyperband (in mlr3hyperband). bbotk is the base package of mlr3tuning, mlr3fselect and miesmuschel.

Author(s)

Maintainer: Marc Becker <marcbecker@posteo.de> (ORCID)

Authors:

• Jakob Richter <jakob1richter@gmail.com> (ORCID)
• Michel Lang <michellang@gmail.com> (ORCID)
• Bernd Bischl <bernd_bischl@gmx.net> (ORCID)
• Martin Binder <martin.binder@mail.com>

Other contributors:

• Olaf Mersmann <olafm@statistik.tu-dortmund.de> [contributor]
See Also

Useful links:

- [https://bbotk.mlr-org.com](https://bbotk.mlr-org.com)
- [https://github.com/mlr-org/bbotk](https://github.com/mlr-org/bbotk)

---

**Archive**

*Logging object for objective function evaluations*

---

**Description**

Container around a `data.table::data.table` which stores all performed function calls of the Objective.

**S3 Methods**

- `as.data.table(archive)`
  
  `Archive -> data.table::data.table()`
  
  Returns a tabular view of all performed function calls of the Objective. The `x_domain` column is unnested to separate columns.

**Public fields**

- `search_space (paradox::ParamSet)`
  
  Search space of objective.

- `codomain (Codomain)`
  
  Codomain of objective function.

- `start_time (POSIXct)`
  
  Time stamp of when the optimization started. The time is set by the Optimizer.

- `check_values (logical(1))`
  
  Determines if points and results are checked for validity.

- `data (data.table::data.table)`
  
  Contains all performed Objective function calls.

- `data_extra (named list)`
  
  Data created by specific Optimizers that does not relate to any individual function evaluation and can therefore not be held in `$data`. Every optimizer should create and refer to its own entry in this list, named by its `class()`.

**Active bindings**

- `n_evals (integer(1))`
  
  Number of evaluations stored in the archive.

- `n_batch (integer(1))`
  
  Number of batches stored in the archive.
cols_x (character())
Column names of search space parameters.

cols_y (character())
Column names of codomain target parameters.

Methods

Public methods:
- Archive$new()
- Archive$add_evals()
- Archive$best()
- Archive$nds_selection()
- Archive$format()
- Archive$print()
- Archive$clear()
- Archive$clone()

Method new(): Creates a new instance of this R6 class.

Usage:
Archive$new(search_space, codomain, check_values = TRUE)

Arguments:
search_space (paradox::ParamSet)
Specifies the search space for the Optimizer. The paradox::ParamSet describes either a
subset of the domain of the Objective or it describes a set of parameters together with
a trafo function that transforms values from the search space to values of the domain.
Depending on the context, this value defaults to the domain of the objective.
codomain (paradox::ParamSet)
Specifies codomain of function. Most importantly the tags of each output “Parameter” de-
fine whether it should be minimized or maximized. The default is to minimize each com-
ponent.
check_values (logical(1))
Should x-values that are added to the archive be checked for validity? Search space that is
logged into archive.

Method add_evals(): Adds function evaluations to the archive table.

Usage:
Archive$add_evals(xdt, xss_trafoed = NULL, ydt)

Arguments:
xdt (data.table::data.table())
Set of untransformed points / points from the search space. One point per row, e.g. data.table(x1
= c(1, 3), x2 = c(2, 4)). Column names have to match ids of the search_space. How-
ever, xdt can contain additional columns.
xss_trafoed (list())
Transformed point(s) in the domain space.
ydt (data.table::data.table())
  Optimal outcome.

**Method best()**: Returns the best scoring evaluation(s). For single-crit optimization, the solution that minimizes / maximizes the objective function. For multi-crit optimization, the Pareto set / front.

*Usage:*
Archive$best(batch = NULL, n_select = 1)

*Arguments:*
  - batch (integer())
    The batch number(s) to limit the best results to. Default is all batches.
  - n_select (integer(1L))
    Amount of points to select. Ignored for multi-crit optimization.

*Returns*: data.table::data.table()

**Method nds_selection()**: Calculate best points w.r.t. non dominated sorting with hypervolume contribution.

*Usage:*
Archive$nds_selection(batch = NULL, n_select = 1, ref_point = NULL)

*Arguments:*
  - batch (integer())
    The batch number(s) to limit the best points to. Default is all batches.
  - n_select (integer(1L))
    Amount of points to select.
  - ref_point (numeric())
    Reference point for hypervolume.

*Returns*: data.table::data.table()

**Method format()**: Helper for print outputs.

*Usage:*
Archive$format()

**Method print()**: Printer.

*Usage:*
Archive$print()

*Arguments:*
  ... (ignored).

**Method clear()**: Clear all evaluation results from archive.

*Usage:*
Archive$clear()

**Method clone()**: The objects of this class are cloneable with this method.

*Usage:*
Archive$clone(deep = FALSE)

*Arguments:*
  deep Whether to make a deep clone.
ArchiveBest

Minimal logging object for objective function evaluations

Description

The ArchiveBest stores no data but records the best scoring evaluation passed to $add_evals()$. The Archive API is fully implemented but many parameters are ignored and some methods do nothing. The archive still works with TerminatorClockTime, TerminatorEvals, TerminatorNone and TerminatorEvals.

Super class

bbotk::Archive -> ArchiveBest

Active bindings

n_evals (integer(1))
   Number of evaluations stored in the archive.

n_batch (integer(1))
   Number of batches stored in the archive.

Methods

Public methods:

• ArchiveBest$new()
• ArchiveBest$add_evals()
• ArchiveBest$best()
• ArchiveBest$clone()

Method new(): Creates a new instance of this R6 class.

Usage:

ArchiveBest$new(search_space, codomain, check_values = FALSE)

Arguments:

search_space (paradox::ParamSet)
   Specifies the search space for the Optimizer. The paradox::ParamSet describes either a subset of the domain of the Objective or it describes a set of parameters together with a trafo function that transforms values from the search space to values of the domain. Depending on the context, this value defaults to the domain of the objective.

codomain (paradox::ParamSet)
   Specifies codomain of function. Most importantly the tags of each output “Parameter” define whether it should be minimized or maximized. The default is to minimize each component.

check_values (logical(1))
   ignored.
Method `add_evals()`: Stores the best result in `ydt`.

Usage:
```r
ArchiveBest$add_evals(xdt, xss_trafoed = NULL, ydt)
```
Arguments:
- `xdt` (`data.table::data.table()`) Set of untransformed points / points from the search space. One point per row, e.g. `data.table(x1 = c(1, 3), x2 = c(2, 4))`. Column names have to match ids of the search_space. However, `xdt` can contain additional columns.
- `xss_trafoed` (list()) Transformed point(s) in the domain space.
- `ydt` (`data.table::data.table()`) Optimal outcome.

Method `best()`: Returns the best scoring evaluation. For single-crit optimization, the solution that minimizes / maximizes the objective function. For multi-crit optimization, the Pareto set / front.

Usage:
```r
ArchiveBest$best(m = NULL)
```
Arguments:
- `m` (integer()) ignored.

Returns: `data.table::data.table()`

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
```r
ArchiveBest$clone(deep = FALSE)
```
Arguments:
- `deep` Whether to make a deep clone.

---

bботк.backup | **Backup Archive Callback**
---

**Description**

This `CallbackOptimization` writes the `Archive` after each batch to disk.

**Examples**

```r
clbk("bботк.backup", path = "backup.rds")
```
**bb_optimize**  
*Black-Box Optimization*

**Description**

This function optimizes a function or Objective with a given method.

**Usage**

```r
bb_optimize(
  x,
  method = "random_search",
  max_evals = 1000,
  max_time = NULL,
  ...
)
```

## S3 method for class 'function'
```r
bb_optimize(
  x,
  method = "random_search",
  max_evals = 1000,
  max_time = NULL,
  lower = NULL,
  upper = NULL,
  maximize = FALSE,
  ...
)
```

## S3 method for class 'Objective'
```r
bb_optimize(
  x,
  method = "random_search",
  max_evals = 1000,
  max_time = NULL,
  search_space = NULL,
  ...
)
```

**Arguments**

- **x** (function | Objective).
- **method** (character(1) | Optimizer)  
  Key to retrieve optimizer from mlr_optimizers dictionary or Optimizer.
- **max_evals** (integer(1))  
  Number of allowed evaluations.
```
max_time (integer(1))
Maximum allowed time in seconds.
...
(named list())
Named arguments passed to objective function. Ignored if Objective is optimized.
lower (numeric())
Lower bounds on the parameters. If named, names are used to create the domain.
upper (numeric())
Upper bounds on the parameters.
maximize (logical())
Logical vector used to create the codomain e.g. c(TRUE, FALSE) -> ps(y1 = p_dbl(tags = "maximize"), y2 = pd_dbl(tags = "minimize")). If named, names are used to create the codomain.
search_space (paradox::ParamSet).

Value

list of
- "par" - Best found parameters
- "value" - Optimal outcome
- "instance" - OptimInstanceSingleCrit | OptimInstanceMultiCrit

Note

If both max_evals and max_time are NULL, TerminatorNone is used. This is useful if the Optimizer can terminate itself. If both are given, TerminatorCombo is created and the optimization stops if the time or evaluation budget is exhausted.

Examples

# function and bounds
fun = function(xs) {
  -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10
}
bb_optimize(fun, lower = c(-10, -5), upper = c(10, 5), max_evals = 10)

# function and constant
fun = function(xs, c) {
  -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + c
}
bb_optimize(fun, lower = c(-10, -5), upper = c(10, 5), max_evals = 10, c = 1)

# objective
fun = function(xs) {
  c(z = -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}
```
# define domain and codomain using a `ParamSet` from paradox
domain = ps(x1 = p_dbl(-10, 10), x2 = p_dbl(-5, 5))
codomain = ps(z = p_dbl(tags = "minimize"))
objective = ObjectiveRFun$new(fun, domain, codomain)

bb_optimize(objective, method = "random_search", max_evals = 10)

---

**branin**

**Branin Function**

### Description

Classic 2-D Branin function with noise `branin(x1, x2, noise)` and Branin function with fidelity parameter `branin_wu(x1, x2, fidelity)`.

### Usage

```r
branin(x1, x2, noise = 0)

branin_wu(x1, x2, fidelity)
```

### Arguments

- `x1` (numeric())
- `x2` (numeric())
- `noise` (numeric())
- `fidelity` (numeric())

### Value

numeric()

### Source


### Examples

```r
branin(x1 = 12, x2 = 2, noise = 0.05)
branin_wu(x1 = 12, x2 = 2, fidelity = 1)
```
Description

Specialized `mlr3misc::Callback` for optimization. Callbacks allow to customize the behavior of processes in bbotk. The `callback_optimization()` function creates a `CallbackOptimization`. Predefined callbacks are stored in the dictionary `mlr_callbacks` and can be retrieved with `clbk()`. For more information on optimization callbacks see `callback_optimization()`.

Super class

`mlr3misc::Callback` -> `CallbackOptimization`

Public fields

- `on_optimization_begin (function())`
  Stage called at the beginning of the optimization. Called in `Optimizer$optimize()`.
- `on_optimizer_before_eval (function())`
  Stage called after the optimizer proposes points. Called in `OptimInstance$eval_batch()`.
- `on_optimizer_after_eval (function())`
  Stage called after points are evaluated. Called in `OptimInstance$eval_batch()`.
- `on_result (function())`
  Stage called after result are written. Called in `OptimInstance$assign_result()`.
- `on_optimization_end (function())`
  Stage called at the end of the optimization. Called in `Optimizer$optimize()`.

Methods

Public methods:

- `CallbackOptimization$clone()

Method clone(): The objects of this class are cloneable with this method.

Usage:

`CallbackOptimization$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

Examples

```r
# write archive to disk
callback_optimization("bbotk.backup",
   on_optimization_end = function(callback, context) {
      saveRDS(context$instance$archive, "archive.rds")
   }
)
```
callback_optimization

Create Optimization Callback

Description

Function to create a CallbackOptimization.

Optimization callbacks can be called from different stages of optimization process. The stages are prefixed with on_*.

Start Optimization
- on_optimization_begin
Start Optimizer Batch
- on_optimizer_before_eval
- on_optimizer_after_eval
End Optimizer Batch
- on_result
- on_optimization_end
End Optimization

See also the section on parameters for more information on the stages. A optimization callback works with ContextOptimization.

Usage

callback_optimization(
  id, label = NA_character_,
  man = NA_character_,
  on_optimization_begin = NULL,
  on_optimizer_before_eval = NULL,
  on_optimizer_after_eval = NULL,
  on_result = NULL,
  on_optimization_end = NULL,
  fields = list()
)

Arguments

id (character(1))
Identifier for the new instance.

label (character(1))
Label for the new instance.

man (character(1))
String in the format [pkg]:[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().
on_optimization_begin
  (function())
  Stage called at the beginning of the optimization. Called in Optimizer$optimize().
  The functions must have two arguments named callback and context.

on_optimizer_before_eval
  (function())
  Stage called after the optimizer proposes points. Called in OptimInstance$eval_batch().
  The functions must have two arguments named callback and context.

on_optimizer_after_eval
  (function())
  Stage called after points are evaluated. Called in OptimInstance$eval_batch().
  The functions must have two arguments named callback and context.

on_result
  (function())
  Stage called after result are written. Called in OptimInstance$assign_result().
  The functions must have two arguments named callback and context.

on_optimization_end
  (function())
  Stage called at the end of the optimization. Called in Optimizer$optimize().
  The functions must have two arguments named callback and context.

fields
  (list of any)
  List of additional fields.

Details

A callback can write data to its state ($state), e.g. settings that affect the callback itself. The ContextOptimization allows to modify the instance, archive, optimizer and final result.

Examples

# write archive to disk
callback_optimization("bbotk.backup",
  on_optimization_end = function(callback, context) {
    saveRDS(context$instance$archive, "archive.rds")
  }
)

Codomain

Codomain of Function

Description

A set of Param objects defining the codomain of a function. The parameter set must contain at least one target parameter tagged with "minimize" or "maximize". The codomain may contain extra parameters which are ignored when calling the Archive methods $best(), $nds_selection() and $cols_y. This class is usually constructed internally from a paradox::ParamSet when Objective is initialized.
Super class

\[ \text{paradox::ParamSet} \rightarrow \text{Codomain} \]

Active bindings

- **is_target** (named logical())
  - Position is TRUE for target Params.
- **target_length** (integer())
  - Returns number of target Params.
- **target_ids** (character())
  - Number of contained target Params.
- **target_tags** (named list() of character())
  - Tags of target Params.
- **maximization_to_minimization** (integer())
  - Returns a numeric vector with values -1 and 1. Multiply with the outcome of a maximization problem to turn it into a minimization problem.

Methods

**Public methods:**

- **Codomain$new()**
- **Codomain$clone()**

**Method new():** Creates a new instance of this R6 class.

**Usage:**

```r
Codomain$new(params = named_list())
```

**Arguments:**

- **params** (list())
  - List of Param, named with their respective ID. Parameters are cloned.

**Method clone():** The objects of this class are cloneable with this method.

**Usage:**

```r
Codomain$clone(deep = FALSE)
```

**Arguments:**

- **deep** Whether to make a deep clone.

Examples

```r
# define objective function
fun = function(xs) {
  c(y = -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
}

# set domain
domain = ps(
```
\[ x_1 = \text{p\_dbl}(-10, 10), \]
\[ x_2 = \text{p\_dbl}(-5, 5) \]

\)

\)

# set codomain

codomain = \text{ps}(

\)

# create Objective object

objective = \text{ObjectiveRFun}\$\text{new}(

\)

ContextOptimization  

Optimization Context

Description

The ContextOptimization allows mlr3misc::Callbacks to access and modify data while optimization. See section on active bindings for a list of modifiable objects. See \text{callback\_optimization()} for a list of stages which access ContextOptimization.

Super class

mlr3misc::Context -> ContextOptimization

Public fields

instance (OptimInstance).

optimizer (Optimizer).

Active bindings

\[ \text{xdt} \ (\text{data\_table::data\_table}) \]

The points of the latest batch. Contains the values in the search space i.e. transformations are not yet applied.

\[ \text{result} \ (\text{data\_table::data\_table}) \]

The result of the optimization.
Methods

Public methods:

• ContextOptimization$new()
• ContextOptimization$clone()

Method new(): Creates a new instance of this R6 class.

Usage:
ContextOptimization$new(instance, optimizer)

Arguments:
instance (OptimInstance).
optimizer (Optimizer).

Method clone(): The objects of this class are cloneable with this method.

Usage:
ContextOptimization$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

is_dominated Calculate which points are dominated

Description

Returns which points from a set are dominated by another point in the set.

Usage

is_dominated(ymat)

Arguments

ymat (matrix())
A numeric matrix. Each column (!) contains one point.
mlr_optimizers

Dictionary of Optimizer

Description

A simple mlr3misc::Dictionary storing objects of class Optimizer. Each optimizer has an associated help page, see mlr_optimizer_[id].

This dictionary can get populated with additional optimizer by add-on packages.

For a more convenient way to retrieve and construct optimizer, see opt()/opts().

Format

R6::R6Class object inheriting from mlr3misc::Dictionary.

Methods

See mlr3misc::Dictionary.

S3 methods

- as.data.table(dict, ..., objects = FALSE)
  
  Returns a data.table::data.table() with fields "key", "label", "param_classes", "properties" and "packages" as columns. If objects is set to TRUE, the constructed objects are returned in the list column named object.

See Also

Sugar functions: opt(), opts()

Examples

as.data.table(mlr_optimizers)
mlr_optimizers$get("random_search")
opt("random_search")

mlr_optimizers_cmaes

Optimization via Covariance Matrix Adaptation Evolution Strategy

Description

OptimizerCmaes class that implements CMA-ES. Calls adagio::pureCMAES() from package adagio. The algorithm is typically applied to search space dimensions between three and fifty. Lower search space dimensions might crash.
Dictionary

This Optimizer can be instantiated via the dictionary mlr_optimizers or with the associated sugar function opt():

```r
mlr_optimizers$get("cmaes")
opt("cmaes")
```

Parameters

- `sigma` numeric(1)
- `start_values` character(1)
  
  Create random start values or based on center of search space? In the latter case, it is the center of the parameters before a trafo is applied.

For the meaning of the control parameters, see `adagio::pureCMAES()`. Note that we have removed all control parameters which refer to the termination of the algorithm and where our terminators allow to obtain the same behavior.

Progress Bars

`$optimize()` supports progress bars via the package `progressr` combined with a Terminator. Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package `progress` as backend; enable with `progressr::handlers("progress")`.

Super class

- bbotk::Optimizer -> OptimizerCmaes

Methods

Public methods:

- `OptimizerCmaes$new()`
- `OptimizerCmaes$clone()`

Method `new()`: Creates a new instance of this R6 class.

Usage:

```r
OptimizerCmaes$new()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```r
OptimizerCmaes$clone(deep = FALSE)
```

Arguments:

- `deep` Whether to make a deep clone.
Examples

```r
if (requireNamespace("adagio")) {
    search_space = domain = ps(
        x1 = p_dbl(-10, 10),
        x2 = p_dbl(-5, 5)
    )

    codomain = ps(y = p_dbl(tags = "maximize"))

    objective_function = function(xs) {
        c(y = -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
    }

    objective = ObjectiveRFun$new(
        fun = objective_function,
        domain = domain,
        codomain = codomain)

    instance = OptimInstanceSingleCrit$new(
        objective = objective,
        search_space = search_space,
        terminator = trm("evals", n_evals = 10))

    optimizer = opt("cmaes")

    # modifies the instance by reference
    optimizer$optimize(instance)

    # returns best scoring evaluation
    instance$result

    # allows access of data.table of full path of all evaluations
    as.data.table(instance$archive$data)
}
```

---

mlr_optimizers_design_points

**Optimization via Design Points**

---

**Description**

OptimizerDesignPoints class that implements optimization w.r.t. fixed design points. We simply search over a set of points fully specified by the user. The points in the design are evaluated in order as given.

In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size batch_size. Larger batches mean we can parallelize more, smaller batches imply a more fine-grained checking of termination criteria.
Dictionary

This Optimizer can be instantiated via the dictionary `mlr_optimizers` or with the associated sugar function `opt()`:

```r
mlr_optimizers$get("design_points")
opt("design_points")
```

Parameters

- `batch_size` integer(1)
  - Maximum number of configurations to try in a batch.
- `design` data.table::data.table
  - Design points to try in search, one per row.

Progress Bars

$optimize()` supports progress bars via the package `progressr` combined with a Terminator. Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package `progress` as backend; enable with `progressr::handlers("progress")`.

Super class

`bbotk::Optimizer` -> `OptimizerDesignPoints`

Methods

**Public methods:**

- `OptimizerDesignPoints$new()`
- `OptimizerDesignPoints$clone()`

**Method new():** Creates a new instance of this R6 class.

*Usage:*

```r
OptimizerDesignPoints$new()
```

**Method clone():** The objects of this class are cloneable with this method.

*Usage:*

```r
OptimizerDesignPoints$clone(deep = FALSE)
```

*Arguments:*

- `deep` Whether to make a deep clone.

Examples

```r
library(data.table)
search_space = domain = ps(x = p_dbl(lower = -1, upper = 1))
codomain = ps(y = p_dbl(tags = "minimize"))
objective_function = function(xs) {
```
list(y = as.numeric(xs)^2)

objective = ObjectiveRFun$new(
  fun = objective_function,
  domain = domain,
  codomain = codomain)

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 10))

design = data.table(x = c(0, 1))

optimizer = opt("design_points", design = design)

# Modifies the instance by reference
optimizer$optimize(instance)

# Returns best scoring evaluation
instance$result

# Allows access of data.table of full path of all evaluations
as.data.table(instance$archive)

---

mlr_optimizers_focus_search

*Optimization via Focus Search*

### Description

OptimizerFocusSearch class that implements a Focus Search.

Focus Search starts with evaluating n_points drawn uniformly at random. For 1 to maxit batches, n_points are then drawn uniformly at random and if the best value of a batch outperforms the previous best value over all batches evaluated so far, the search space is shrinked around this new best point prior to the next batch being sampled and evaluated.

For details on the shrinking, see `shrink_ps`.

Depending on the Terminator this procedure simply restarts after maxit is reached.

### Dictionary

This Optimizer can be instantiated via the dictionary `mlr_optimizers` or with the associated sugar function `opt()`:

```r
mlr_optimizers$get("focus_search")
opt("focus_search")
```
Parameters

- **n_points** integer(1)
  Number of points to evaluate in each random search batch.

- **maxit** integer(1)
  Number of random search batches to run.

Progress Bars
$\texttt{optimize()}$ supports progress bars via the package **progressr** combined with a **Terminator**. Simply wrap the function in **\texttt{progressr::with_progress()}** to enable them. We recommend to use package **progress** as backend; enable with **\texttt{progressr::handlers("progress")}**.

Super class

  `bbotk::Optimizer` \rightarrow \texttt{OptimizerFocusSearch}

Methods

Public methods:

- `OptimizerFocusSearch$new()`
- `OptimizerFocusSearch$clone()`

Method **\texttt{new()}:** Creates a new instance of this R6 class.

  Usage:

  \texttt{OptimizerFocusSearch$new()}

Method **\texttt{clone()}:** The objects of this class are cloneable with this method.

  Usage:

  \texttt{OptimizerFocusSearch$clone(deep = \texttt{FALSE})}

  Arguments:

  - \texttt{deep} Whether to make a deep clone.

Examples

```r
search_space = domain = ps(x = p_dbl(lower = -1, upper = 1))
codomain = ps(y = p_dbl(tags = "minimize"))

objective_function = function(xs) {
  list(y = as.numeric(xs)^2)
}

objective = ObjectiveRFun$new(
  fun = objective_function,
  domain = domain,
  codomain = codomain)

instance = OptimInstanceSingleCrit$new(
  domain = search_space,
  codomain = codomain,
  objective = objective,
  control = control()
)
```
OptimizerGenSA class that implements generalized simulated annealing. Calls `GenSA::GenSA()` from package `GenSA`.

This Optimizer can be instantiated via the dictionary `mlr_optimizers` or with the associated sugar function `opt()`:

```r
mlr_optimizers$get("gensa")
opt("gensa")
```

Parameters

- smooth logical(1)
- temperature numeric(1)
- acceptance.param numeric(1)
- verbose logical(1)
- trace.mat logical(1)

For the meaning of the control parameters, see `GenSA::GenSA()`. Note that we have removed all control parameters which refer to the termination of the algorithm and where our terminators allow to obtain the same behavior.

In contrast to the `GenSA::GenSA()` defaults, we set `trace.mat = FALSE`. Note that `GenSA::GenSA()` uses `smooth = TRUE` as a default. In the case of using this optimizer for Hyperparameter Optimization you may want to set `smooth = FALSE`. 
Progress Bars

$optimize()$ supports progress bars via the package \texttt{progressr} combined with a \texttt{Terminator}. Simply wrap the function in \texttt{progressr::with_progress()} to enable them. We recommend to use package \texttt{progress} as backend; enable with \texttt{progressr::handlers("progress")}.

Super class

\texttt{bbotk::Optimizer} $\rightarrow$ \texttt{OptimizerGenSA}

Methods

Public methods:

- \texttt{OptimizerGenSA$new()}
- \texttt{OptimizerGenSA$clone()}

Method \texttt{new()}: Creates a new instance of this \texttt{R6} class.

Usage:

\texttt{OptimizerGenSA$new()}

Method \texttt{clone()}: The objects of this class are cloneable with this method.

Usage:

\texttt{OptimizerGenSA$clone(deep = FALSE)}

Arguments:

depth Whether to make a deep clone.

Source


Examples

```r
if (requireNamespace("GenSA")) {
  search_space = domain = ps(x = p_dbl(lower = -1, upper = 1))
  codomain = ps(y = p_dbl(tags = "minimize"))
  objective_function = function(xs) {
    list(y = as.numeric(xs)^2)
  }
  objective = ObjectiveRFun$new(
    fun = objective_function,
    domain = domain,
    codomain = codomain)
}
```
mlr_optimizers_grid_search

Optimization via Grid Search

Description

OptimizerGridSearch class that implements grid search. The grid is constructed as a Cartesian product over discretized values per parameter, see `paradox::generate_design_grid()`. The points of the grid are evaluated in a random order.

In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size `batch_size`. Larger batches mean we can parallelize more, smaller batches imply a more fine-grained checking of termination criteria.

Dictionary

This Optimizer can be instantiated via the dictionary `mlr_optimizers` or with the associated sugar function `opt()`:

```r
mlr_optimizers$get("grid_search")
```

```r
opt("grid_search")
```

Parameters

- **resolution integer(1)**
  - Resolution of the grid, see `paradox::generate_design_grid()`.

- **param_resolutions named integer()**
  - Resolution per parameter, named by parameter ID, see `paradox::generate_design_grid()`.

- **batch_size integer(1)**
  - Maximum number of points to try in a batch.
Progress Bars

$optimize() supports progress bars via the package \texttt{progressr} combined with a \texttt{Terminator}. Simply wrap the function in \texttt{progressr::with_progress()} to enable them. We recommend to use package \texttt{progress} as backend; enable with \texttt{progressr::handlers(“progress”).}

Super class

\texttt{bbotk::Optimizer} \rightarrow \texttt{OptimizerGridSearch}

Methods

Public methods:

- \texttt{OptimizerGridSearch}\texttt{\textbullet{}new()}
- \texttt{OptimizerGridSearch}\texttt{\textbullet{}}\texttt{clone()}

Method \texttt{new()}: Creates a new instance of this \texttt{R6} class.

Usage:

\texttt{OptimizerGridSearch}\texttt{\textbullet{}new()}

Method \texttt{clone()}: The objects of this class are cloneable with this method.

Usage:

\texttt{OptimizerGridSearch}\texttt{\textbullet{}clone(deep = FALSE)}

Arguments:

depth Whether to make a deep clone.

Examples

\begin{verbatim}
search_space = domain = ps(x = p_dbl(lower = -1, upper = 1))
codomain = ps(y = p_dbl(tags = "minimize"))

objective_function = function(xs) {
  list(y = as.numeric(xs)^2)
}

objective = ObjectiveRFun$new(
  fun = objective_function,
  domain = domain,
  codomain = codomain)

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 10))

optimizer = opt("grid_search")

# modifies the instance by reference
\end{verbatim}
mlr_optimizers_irace

optimizer$optimize(instance)

# returns best scoring evaluation
instance$result

# allows access of data.table of full path of all evaluations
as.data.table(instance$archive$data)

mlr_optimizers_irace  Optimization via Iterated Racing

Description

OptimizerRrace class that implements iterated racing. Calls irace::irace() from package irace.

Parameters

instances list()
  A list of instances where the configurations executed on.
targetRunnerParallel function()
  A function that executes the objective function with a specific parameter configuration and instance. A default function is provided, see section "Target Runner and Instances".

For the meaning of all other parameters, see irace::defaultScenario(). Note that we have removed all control parameters which refer to the termination of the algorithm. Use TerminatorEvals instead. Other terminators do not work with OptimizerIrace.

In contrast to irace::defaultScenario(), we set digits = 15. This represents double parameters with a higher precision and avoids rounding errors.

Target Runner and Instances

The irace package uses a targetRunner script or R function to evaluate a configuration on a particular instance. Usually it is not necessary to specify a targetRunner function when using OptimizerIrace. A default function is used that forwards several configurations and instances to the user defined objective function. As usually, the user defined function has a xs, xss or xdt parameter depending on the used Objective class. For irace, the function needs an additional instances parameter.

fun = function(xs, instances) {
  # function to evaluate configuration in `xs` on instance `instances`
}

Archive

The Archive holds the following additional columns:

- "race" (integer(1))
  Race iteration.
• "step" (integer(1))  
  Step number of race.
• "instance" (integer(1))  
  Identifies instances across races and steps.
• "configuration" (integer(1))  
  Identifies configurations across races and steps.

Result

The optimization result (instance$result) is the best performing elite of the final race. The reported performance is the average performance estimated on all used instances.

Dictionary

This Optimizer can be instantiated via the dictionary `mlr_optimizers` or with the associated sugar function `opt()`:

```r
mlr_optimizers$get("irace")
opt("irace")
```

Progress Bars

$optimize() supports progress bars via the package `progressr` combined with a Terminator. Simply wrap the function in `progressr::with_progress()` to enable them. We recommend to use package `progress` as backend; enable with `progressr::handlers("progress")`.

Super class

`bbotk::Optimizer` -> `OptimizerIrace`

Methods

Public methods:

• `OptimizerIrace$new()`
• `OptimizerIrace$clone()`

Method `new()`: Creates a new instance of this `R6` class.

Usage:

```r
OptimizerIrace$new()
```

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```r
OptimizerIrace$clone(deep = FALSE)
```

Arguments:

deepl Whether to make a deep clone.
Source


Examples

```r
library(data.table)

space = domain = ps(
  x1 = p_dbl(-5, 10),
  x2 = p_dbl(0, 15)
)

codomain = ps(y = p_dbl(tags = "minimize"))

# branin function with noise
# the noise generates different instances of the branin function
# the noise values are passed via the `instances` parameter
fun = function(xdt, instances) {
  ys = branin(xdt[["x1"]], xdt[["x2"]], noise = as.numeric(instances))
  data.table(y = ys)
}

# define objective with instances as a constant
objective = ObjectiveRFunDt$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  constants = ps(instances = p_util())
)

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = space,
  terminator = trm("evals", n_evals = 1000))

# create instances of branin function
instances = rnorm(10, mean = 0, sd = 0.1)

# load optimizer irace and set branin instances
optimizer = opt("irace", instances = instances)

# modifies the instance by reference
optimizer$optimize(instance)

# best scoring configuration
instance$result

# all evaluations
as.data.table(instance$archive)
```
mlr_optimizers_nloptr

Optimization via Non-linear Optimization

Description

OptimizerNLoptr class that implements non-linear optimization. Calls nloptr::nloptr() from package nloptr.

Parameters

- algorithm character(1)
- eval_g_ineq function()
- xtol_rel numeric(1)
- xtol_abs numeric(1)
- ftol_rel numeric(1)
- ftol_abs numeric(1)
- start_values character(1)

Create random start values or based on center of search space? In the latter case, it is the center of the parameters before a trafo is applied.

For the meaning of the control parameters, see nloptr::nloptr() and nloptr::nloptr.print.options().

The termination conditions stopval, maxtime and maxeval of nloptr::nloptr() are deactivated and replaced by the Terminator subclasses. The x and function value tolerance termination conditions (xtol_rel = 10^-4, xtol_abs = rep(0.0, length(x0)), ftol_rel = 0.0 and ftol_abs = 0.0) are still available and implemented with their package defaults. To deactivate these conditions, set them to -1.

Progress Bars

$optimize() supports progress bars via the package progressr combined with a Terminator. Simply wrap the function in progressr::with_progress() to enable them. We recommend to use package progress as backend; enable with progressr::handlers("progress").

Super class

bbotk::Optimizer -> OptimizerNLoptr

Methods

Public methods:

- OptimizerNLoptr$new()
- OptimizerNLoptr$clone()

Method new(): Creates a new instance of this R6 class.

Usage:

OptimizerNLoptr$new()
Method clone(): The objects of this class are cloneable with this method.

Usage:
OptimizerNLoptr$clone(deep = FALSE)

Arguments:
depth Whether to make a deep clone.

Source

Examples

if (requireNamespace("nloptr")) {

  search_space = domain = ps(x = p_dbl(lower = -1, upper = 1))
  codomain = ps(y = p_dbl(tags = "minimize"))
  objective_function = function(xs) {
    list(y = as.numeric(xs)^2)
  }

  objective = ObjectiveRFun$new(
    fun = objective_function,
    domain = domain,
    codomain = codomain)

  # We use the internal termination criterion xtol_rel
  terminator = trm("none")
  instance = OptimInstanceSingleCrit$new(
    objective = objective,
    search_space = search_space,
    terminator = terminator)

  optimizer = opt("nloptr", algorithm = "NLOPT_LN_BOBYQA")

  # Modifies the instance by reference
  optimizer$optimize(instance)

  # Returns best scoring evaluation
  instance$result

  # Allows access of data.table of full path of all evaluations
  as.data.table(instance$archive)
}
Optimizer class that implements a simple Random Search.

In order to support general termination criteria and parallelization, we evaluate points in a batch-fashion of size batch_size. Larger batches mean we can parallelize more, smaller batches imply a more fine-grained checking of termination criteria.

This Optimizer can be instantiated via the dictionary mlr_optimizers or with the associated sugar function opt():

```r
mlr_optimizers$get("random_search")
opt("random_search")
```

Parameters

- batch_size integer(1)
  - Maximum number of points to try in a batch.

Progress Bars

$optimize() supports progress bars via the package progressr combined with a Terminator. Simply wrap the function in progressr::with_progress() to enable them. We recommend to use package progress as backend; enable with progressr::handlers("progress").

Super class

bbotk::Optimizer -> OptimizerRandomSearch

Methods

- **Public methods:**
  - OptimizerRandomSearch$new()
  - OptimizerRandomSearch$clone()

  **Method new():** Creates a new instance of this R6 class.

  **Usage:**

  ```r
  OptimizerRandomSearch$new()
  ```

  **Method clone():** The objects of this class are cloneable with this method.

  **Usage:**

  ```r
  OptimizerRandomSearch$clone(deep = FALSE)
  ```

  **Arguments:**

  deep Whether to make a deep clone.
Source


Examples

```r
search_space = domain = ps(x = p_dbl(lower = -1, upper = 1))

codomain = ps(y = p_dbl(tags = "minimize"))

objective_function = function(xs) {
  list(y = as.numeric(xs)^2)
}

objective = ObjectiveRFun$new(
  fun = objective_function,
  domain = domain,
  codomain = codomain)

instance = OptimInstanceSingleCrit$new(
  objective = objective,
  search_space = search_space,
  terminator = trm("evals", n_evals = 10))

optimizer = opt("random_search")

# modifies the instance by reference
optimizer$optimize(instance)

# returns best scoring evaluation
instance$result

# allows access of data.table of full path of all evaluations
as.data.table(instance$archive$data)
```

---

**mlr_terminators**

**Dictionary of Terminators**

**Description**

A simple `mlr3misc::Dictionary` storing objects of class `Terminator`. Each terminator has an associated help page, see `mlr_terminators_[id]`.

This dictionary can get populated with additional terminators by add-on packages.

For a more convenient way to retrieve and construct terminator, see `trm()`/`trms()`.
**mlr_terminators_clock_time**

**Format**

*R6::R6Class* object inheriting from *mlr3misc::Dictionary*.

**Methods**

See *mlr3misc::Dictionary*.

**S3 methods**

- `as.data.table(dict, ..., objects = FALSE)
  *mlr3misc::Dictionary* -> *data.table::data.table()*
  Returns a *data.table::data.table()* with fields "key", "label", "properties" and "unit" as columns. If `objects` is set to TRUE, the constructed objects are returned in the list column named `object`.

**See Also**

Sugar functions: `trm()`, `trms()`


**Examples**

```r
as.data.table(mlr_terminators)
mlr_terminators$get("evals")
trm("evals", n_evals = 10)
```

---

**mlr_terminators_clock_time**

*Clock Time Terminator*

**Description**

Class to terminate the optimization after a fixed time point has been reached (as reported by *Sys.time()*).

**Dictionary**

This *Terminator* can be instantiated via the dictionary *mlr_terminators* or with the associated sugar function `trm()`:

```r
mlr_terminators$get("clock_time")
trm("clock_time")
```

**Parameters**

- `stop_time` *POSIXct*<1>
  Terminator stops after this point in time.
Super class

bbotk::Terminator -> TerminatorClockTime

Methods

Public methods:

• TerminatorClockTime$new()
• TerminatorClockTime$is_terminated()
• TerminatorClockTime$clone()

Method new(): Creates a new instance of this R6 class.
Usage:
TerminatorClockTime$new()

Method is_terminated(): Is TRUE iff the termination criterion is positive, and FALSE otherwise.
Usage:
TerminatorClockTime$is_terminated(archive)
Arguments:
archive (Archive).
Returns: logical(1).

Method clone(): The objects of this class are cloneable with this method.
Usage:
TerminatorClockTime$clone(deep = FALSE)
Arguments:
depth Whether to make a deep clone.

See Also


Examples

stop_time = as.POSIXct("2030-01-01 00:00:00")
trm("clock_time", stop_time = stop_time)
**Combine Terminators**

**Description**

This class takes multiple Terminators and terminates as soon as one or all of the included terminators are positive.

**Dictionary**

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function `trm()`:

```r
mlr_terminators$get("combo")
trm("combo")
```

**Parameters**

- any logical(1)
  
  Terminate iff any included terminator is positive? (not all). Default is TRUE.

**Super class**

`bbotk::Terminator` -> TerminatorCombo

**Public fields**

- terminators (list())
  
  List of objects of class Terminator.

**Methods**

**Public methods:**

- `TerminatorCombo$new()`
- `TerminatorCombo$is_terminated()`
- `TerminatorCombo$print()`
- `TerminatorCombo$remaining_time()`
- `TerminatorCombo$status_long()`
- `TerminatorCombo$clone()`

**Method new():** Creates a new instance of this R6 class.

**Usage:**

```r
TerminatorCombo$new(terminators = list(TerminatorNone$new()))
```

**Arguments:**

- terminators (list())
  
  List of objects of class Terminator.
Method `is_terminated()`: Is TRUE iff the termination criterion is positive, and FALSE otherwise.

Usage:
```
TerminatorCombo$is_terminated(archive)
```

Arguments:
archive (Archive).

Returns: logical(1).

Method `print()`: Printer.

Usage:
```
TerminatorCombo$print(...)  
```

Arguments:
... (ignored).

Method `remaining_time()`: Returns the remaining runtime in seconds. If any = TRUE, the remaining runtime is determined by the time-based terminator with the shortest time remaining. If non-time-based terminators are used and any = FALSE, the the remaining runtime is always Inf.

Usage:
```
TerminatorCombo$remaining_time(archive)
```

Arguments:
archive (Archive).

Returns: integer(1).

Method `status_long()`: Returns max_steps and current_steps for each terminator.

Usage:
```
TerminatorCombo$status_long(archive)
```

Arguments:
archive (Archive).

Returns: data.table::data.table.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
```
TerminatorCombo$clone(deep = FALSE)
```

Arguments:
deep Whether to make a deep clone.

See Also

Examples

```
trm("combo",
    list(trm("clock_time", stop_time = Sys.time() + 60),
         trm("evals", n_evals = 10)), any = FALSE
)
```

**mlr_terminators_evals**  
Terminator that stops after a number of evaluations

**Description**

Class to terminate the optimization depending on the number of evaluations. An evaluation is defined by one resampling of a parameter value. The total number of evaluations $B$ is defined as

$$B = n_{\text{evals}} + k \times D$$

where $D$ is the dimension of the search space.

**Dictionary**

This Terminator can be instantiated via the dictionary `mlr_terminators` or with the associated sugar function `trm()`:

```
mlr_terminators$get("evals")
trm("evals")
```

**Parameters**

- `n_evals` integer(1)
  - See formula above. Default is 100.
- `k` integer(1)
  - See formula above. Default is 0.

**Super class**

`bbotk::Terminator` -> `TerminatorEvals`

**Methods**

**Public methods:**

- `TerminatorEvals$new()`
- `TerminatorEvals$is_terminated()`
- `TerminatorEvals$clone()`

**Method** `new()`: Creates a new instance of this R6 class.

Usage:
TerminatorEvals$new()

**Method** `is_terminated()`: Is TRUE iff the termination criterion is positive, and FALSE otherwise.

*Usage:*
TerminatorEvals$is_terminated(archive)

*Arguments:*
archive (Archive).

*Returns:* logical(1).

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*
TerminatorEvals$clone(deep = FALSE)

*Arguments:*
deep Whether to make a deep clone.

See Also


Examples

TerminatorEvals$new()

# 5 evaluations in total
trm("evals", n_evals = 5)

# 3 * [dimension of search space] evaluations in total
trm("evals", n_evals = 0, k = 3)

# (3 * [dimension of search space] + 1) evaluations in total
trm("evals", n_evals = 1, k = 3)

---

None Terminator

**Description**

Mainly useful for optimization algorithms where the stopping is inherently controlled by the algorithm itself (e.g. OptimizerGridSearch).
Dictionary

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function trm:

```r
dl = mlr_terminators$get("none")
trm("none")
```

Super class

`bbotk::Terminator` -> `TerminatorNone`

Methods

Public methods:

- `TerminatorNone$new()`
- `TerminatorNone$is_terminated()`
- `TerminatorNone$clone()`

Method `new()`: Creates a new instance of this R6 class.

Usage:

```r
TerminatorNone$new()
```

Method `is_terminated()`: Is TRUE iff the termination criterion is positive, and FALSE otherwise.

Usage:

```r
TerminatorNone$is_terminated(archive)
```

Arguments:

archive (Archive).

Returns: logical(1).

Method `clone()`: The objects of this class are cloneable with this method.

Usage:

```r
TerminatorNone$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

See Also

**mlr_terminators_perf_reached**

*Performance Level Terminator*

**Description**

Class to terminate the optimization after a performance level has been hit.

**Dictionary**

This Terminator can be instantiated via the dictionary `mlr_terminators` or with the associated sugar function `trm()`:

```r
mlr_terminators$get("perf_reached")
trm("perf_reached")
```

**Parameters**

`level numeric(1)`

Performance level that needs to be reached. Default is 0. Terminates if the performance exceeds (respective measure has to be maximized) or falls below (respective measure has to be minimized) this value.

**Super class**

`bbotk::Terminator` -> `TerminatorPerfReached`

**Methods**

**Public methods:**

- `TerminatorPerfReached$new()`
- `TerminatorPerfReached$is_terminated()`
- `TerminatorPerfReached$clone()`

**Method** `new()`: Creates a new instance of this R6 class.

*Usage:*

`TerminatorPerfReached$new()`

**Method** `is_terminated()`: Is TRUE iff the termination criterion is positive, and FALSE otherwise.

*Usage:*

`TerminatorPerfReached$is_terminated(archive)`

*Arguments:*

- `archive` (Archive).

*Returns:*

logical(1).
**Method** clone(): The objects of this class are cloneable with this method.

**Usage:**
TerminatorPerfReached$clone(deep = FALSE)

**Arguments:**
deep Whether to make a deep clone.

See Also


Examples

TerminatorPerfReached$new()
trm("perf_reached")

---

**mlr_terminators_run_time**

*Run Time Terminator*

**Description**

Class to terminate the optimization after the optimization process took a number of seconds on the clock.

**Dictionary**

This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function trm():

mlr_terminators$get("run_time")
trm("run_time")

**Parameters**

secs numeric(1)
  Maximum allowed time, in seconds, default is 100.

**Super class**

bbotk::Terminator -> TerminatorRunTime
Methods

Public methods:

• TerminatorRunTime$new()
• TerminatorRunTime$is_terminated()
• TerminatorRunTime$clone()

Method new(): Creates a new instance of this R6 class.

Usage:
TerminatorRunTime$new()

Method is_terminated(): Is TRUE iff the termination criterion is positive, and FALSE otherwise.

Usage:
TerminatorRunTime$is_terminated(archive)

Arguments:
archive (Archive).

Returns: logical(1).

Method clone(): The objects of this class are cloneable with this method.

Usage:
TerminatorRunTime$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

Note

This terminator only works if archive$start_time is set. This is usually done by the Optimizer.

See Also


Examples

trm("run_time", secs = 1800)
Terminator that stops when optimization does not improve

Description
Class to terminate the optimization after the performance stagnates, i.e. does not improve more than threshold over the last iters iterations.

Dictionary
This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function trm():

```r
mlr_terminators$get("stagnation")
trm("stagnation")
```

Parameters
- **iters** integer(1)
  Number of iterations to evaluate the performance improvement on, default is 10.
- **threshold** numeric(1)
  If the improvement is less than threshold, optimization is stopped, default is 0.

Super class
bbotk::Terminator -&gt; TerminatorStagnation

Methods
Public methods:
- `TerminatorStagnation$new()`
- `TerminatorStagnation$is_terminated()`
- `TerminatorStagnation$clone()`

Method `new()`: Creates a new instance of this R6 class.

Usage:
`TerminatorStagnation$new()`

Method `is_terminated()`: Is TRUE iff the termination criterion is positive, and FALSE otherwise.

Usage:
`TerminatorStagnation$is_terminated(archive)`

Arguments:
archive (Archive).
Returns: logical(1).

Method clone(): The objects of this class are cloneable with this method.

Usage:
TerminatorStagnation$clone(deep = FALSE)

Arguments:
depth Whether to make a deep clone.

See Also

Examples
TerminatorStagnation$new()
trm("stagnation", iters = 5, threshold = 1e-5)

mlr_terminators_stagnation_batch
Terminator that stops when optimization does not improve

Description
Class to terminate the optimization after the performance stagnates, i.e. does not improve more than threshold over the last n batches.

Dictionary
This Terminator can be instantiated via the dictionary mlr_terminators or with the associated sugar function trm():

mlr_terminators$get("stagnation_batch")
trm("stagnation_batch")

Parameters
n integer(1)
Number of batches to evaluate the performance improvement on, default is 1.

threshold numeric(1)
If the improvement is less than threshold, optimization is stopped, default is 0.

Super class
bbotk::Terminator -> TerminatorStagnationBatch
Methods

Public methods:

- `TerminatorStagnationBatch$new()`
- `TerminatorStagnationBatch$is_terminated()`
- `TerminatorStagnationBatch$clone()`

**Method new()**: Creates a new instance of this R6 class.

*Usage*

TerminatorStagnationBatch$new()

**Method is_terminated()**: Is TRUE iff the termination criterion is positive, and FALSE otherwise.

*Usage*

TerminatorStagnationBatch$is_terminated(archive)

*Arguments*

archive (Archive).

*Returns*: logical(1).

**Method clone()**: The objects of this class are cloneable with this method.

*Usage*

TerminatorStagnationBatch$clone(deep = FALSE)

*Arguments*

dee[p] Whether to make a deep clone.

See Also


Examples

TerminatorStagnationBatch$new()

trm("stagnation_batch", n = 1, threshold = 1e-5)
**Objective**

*Objective function with domain and co-domain*

**Description**

Describes a black-box objective function that maps an arbitrary domain to a numerical codomain.

**Technical details**

Objective objects can have the following properties: "noisy", "deterministic", "single-crit" and "multi-crit".

**Public fields**

- `id` (character(1))
- `properties` (character(1))
- `domain` (paradox::ParamSet)
  - Specifies domain of function, hence its input parameters, their types and ranges.
- `codomain` (paradox::ParamSet)
  - Specifies codomain of function, hence its feasible values.
- `constants` (paradox::ParamSet)
  - Changeable constants or parameters that are not subject to tuning can be stored and accessed here. Set constant values are passed to `.eval()` and `.eval_many()` as named arguments.
- `check_values` (logical(1))

**Active bindings**

- `xdim` (integer(1))
  - Dimension of domain.
- `ydim` (integer(1))
  - Dimension of codomain.

**Methods**

**Public methods:**

- `Objective$new()`  
- `Objective$format()`  
- `Objective$print()`  
- `Objective$eval()`  
- `Objective$eval_many()`  
- `Objective$eval_dt()`  
- `Objective$clone()`  

**Method** `new()`: Creates a new instance of this `R6` class.
Objective

Usage:
Objective$new(
  id = "f",
  properties = character(),
  domain,
  codomain = ps(y = p_dbl(tags = "minimize")),
  constants = ps(),
  check_values = TRUE
)

Arguments:
id (character(1)).
properties (character()).
domain (paradox::ParamSet)
  Specifies domain of function. The paradox::ParamSet should describe all possible input
  parameters of the objective function. This includes their id, their types and the possible
  range.
codomain (paradox::ParamSet)
  Specifies codomain of function. Most importantly the tags of each output "Parameter" de-
  fine whether it should be minimized or maximized. The default is to minimize each com-
  ponent.
constants (paradox::ParamSet)
  Changeable constants or parameters that are not subject to tuning can be stored and accessed
  here.
check_values (logical(1))
  Should points before the evaluation and the results be checked for validity?

Method format(): Helper for print outputs.
Usage:
Objective$format()
Returns: character().

Method print(): Print method.
Usage:
Objective$print()
Returns: character().

Method eval(): Evaluates a single input value on the objective function. If check_values =
TRUE, the validity of the point as well as the validity of the result is checked.
Usage:
Objective$eval(xs)
Arguments:
x (list())
  A list that contains a single x value, e.g. list(x1 = 1, x2 = 2).
Returns:  list() that contains the result of the evaluation, e.g. list(y = 1). The list can
  also contain additional named entries that will be stored in the archive if called through the
  OptimInstance. These extra entries are referred to as extras.
Method `eval_many()`: Evaluates multiple input values on the objective function. If `check_values = TRUE`, the validity of the points as well as the validity of the results are checked. `bbotk` does not take care of parallelization. If the function should make use of parallel computing, it has to be implemented by deriving from this class and overwriting this function.

Usage:
Objective$eval_many(xss)

Arguments:

`xss` (list())
A list of lists that contains multiple x values, e.g. `list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4))`.

Returns: data.table::data.table() that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. `data.table(y1 = 1:2, y2 = 3:4)`. It may also contain additional columns that will be stored in the archive if called through the `OptimInstance`. These extra columns are referred to as `extras`.

Method `eval_dt()`: Evaluates multiple input values on the objective function

Usage:
Objective$eval_dt(xdt)

Arguments:

`xdt` (data.table::data.table())
Set of untransformed points / points from the search space. One point per row, e.g. `data.table(x1 = c(1, 3), x2 = c(2, 4))`. Column names have to match ids of the search_space. However, xdt can contain additional columns.

Returns: data.table::data.table() that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. `data.table(y1 = 1:2, y2 = 3:4)`.

Method `clone()`: The objects of this class are cloneable with this method.

Usage:
Objective$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

---

**ObjectiveRFun**

**Objective interface with custom R function**

Description

Objective interface where the user can pass a custom R function that expects a list as input. If the return of the function is unnamed, it is named with the ids of the codomain.

Super class

`bbotk::Objective` -> `ObjectiveRFun`
### ObjectiveRFun

**Active bindings**

```r
fun (function)
Objective function.
```

**Methods**

**Public methods:**

- `ObjectiveRFun$new()`
- `ObjectiveRFun$eval()`
- `ObjectiveRFun$clone()`

**Method `new()`:** Creates a new instance of this R6 class.

**Usage:**

```r
ObjectiveRFun$new(
  fun,  # R function that encodes objective and expects a list with the input for a single point (e.g. list(x1 = 1, x2 = 2)) and returns the result either as a numeric vector or a list (e.g. list(y = 3)).
  domain,  # specifies domain of function. The paradox::ParamSet should describe all possible input parameters of the objective function. This includes their id, their types and the possible range.
  codomain = NULL,  # specifies codomain of function. Most importantly the tags of each output "Parameter" define whether it should be minimized or maximized. The default is to minimize each component.
  id = "function",  # (character(1)).
  properties = character(),  # (character()).
  constants = ps(),  # Changeable constants or parameters that are not subject to tuning can be stored and accessed here.
  check_values = TRUE
)
```

**Arguments:**

- `fun (function)`
- `domain (paradox::ParamSet)`
- `codomain (paradox::ParamSet)`
- `id (character(1))`
- `properties (character())`
- `constants (paradox::ParamSet)`
- `check_values (logical(1))`

**Method `eval()`:** Evaluates input value(s) on the objective function. Calls the R function supplied by the user.

**Usage:**
ObjectiveRFun$eval(xs)

*Arguments:*
xs  Input values.

**Method clone():** The objects of this class are cloneable with this method.

*Usage:*
ObjectiveRFun$clone(deep = FALSE)

*Arguments:*
deep  Whether to make a deep clone.

**Examples**

```r
# define objective function
fun = function(xs) {
  -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10
}

# set domain
domain = ps(
  x1 = p_dbl(-10, 10),
  x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(y = p_dbl(tags = "maximize"))

# create Objective object
obfun = ObjectiveRFun$new(
  fun = fun,
  domain = domain,
  codomain = codomain,
  properties = "deterministic"
)
```

---

**ObjectiveRFunDt**  
*Objective interface for basic R functions.*

**Description**

Objective interface where user can pass an R function that works on an `data.table()`.

**Super class**

`bbotk::Objective` -> `ObjectiveRFunDt`

**Active bindings**

fun (function)  
Objective function.
Methods

Public methods:

- ObjectiveRFunDt$new()
- ObjectiveRFunDt$eval_many()
- ObjectiveRFunDt$eval_dt()
- ObjectiveRFunDt$clone()

Method `new()`: Creates a new instance of this R6 class.

Usage:

```r
ObjectiveRFunDt$new(
  fun,
  domain,
  codomain = NULL,
  id = "function",
  properties = character(),
  constants = ps(),
  check_values = TRUE
)
```

Arguments:

- `fun` (function)
  
  R function that encodes objective and expects an `data.table()` as input whereas each point is represented by one row.

- `domain` (paradox::ParamSet)
  
  Specifies domain of function. The paradox::ParamSet should describe all possible input parameters of the objective function. This includes their id, their types and the possible range.

- `codomain` (paradox::ParamSet)
  
  Specifies codomain of function. Most importantly the tags of each output "Parameter" define whether it should be minimized or maximized. The default is to minimize each component.

- `id` (character(1)).

- `properties` (character()).

- `constants` (paradox::ParamSet)
  
  Changeable constants or parameters that are not subject to tuning can be stored and accessed here.

- `check_values` (logical(1))
  
  Should points before the evaluation and the results be checked for validity?

Method `eval_many()`: Evaluates multiple input values received as a list, converted to a `data.table()` on the objective function. Missing columns in `xss` are filled with NAs in `xdt`.

Usage:

```r
ObjectiveRFunDt$eval_many(xss)
```

Arguments:

- `xss` (list())
  
  A list of lists that contains multiple x values, e.g. `list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4))`. 
**ObjectiveRFunMany**

**Returns:** `data.table::data.table()` that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. `data.table(y = 1:2)` or `data.table(y1 = 1:2, y2 = 3:4).

**Method** `eval_dt()`: Evaluates multiple input values on the objective function supplied by the user.

**Usage:**
`ObjectiveRFunDt$eval_dt(xdt)`

**Arguments:**
- **xdt** (`data.table::data.table()`): Set of untransformed points / points from the `search space`. One point per row, e.g. `data.table(x1 = c(1, 3), x2 = c(2, 4))`. Column names have to match ids of the `search_space`. However, `xdt` can contain additional columns.

**Returns:** `data.table::data.table()` that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. `data.table(y = 1:2)` or `data.table(y1 = 1:2, y2 = 3:4).

**Method** `clone()`: The objects of this class are cloneable with this method.

**Usage:**
`ObjectiveRFunDt$clone(deep = FALSE)`

**Arguments:**
- **deep** (`logical()`) Whether to make a deep clone.

---

**ObjectiveRFunMany**

**Objective Interface with Custom R Function**

**Description**

Objective interface where the user can pass a custom R function that expects a list of configurations as input. If the return of the function is unnamed, it is named with the ids of the codomain.

**Super class**

`bbotk::Objective` -> `ObjectiveRFunMany`

**Active bindings**

- **fun** (`function`): Objective function.
Methods

Public methods:

- `ObjectiveRFunMany$new()`
- `ObjectiveRFunMany$eval_many()`
- `ObjectiveRFunMany$clone()`

Method `new()`: Creates a new instance of this R6 class.

Usage:

```r
ObjectiveRFunMany$new(
  fun, 
  domain, 
  codomain = NULL, 
  id = "function", 
  properties = character(), 
  constants = ps(), 
  check_values = TRUE 
)
```

Arguments:

- `fun` (function)
  - R function that encodes objective and expects a list of lists that contains multiple x values, e.g. `list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4))`. The function must return a `data.table::data.table()` that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. `data.table(y = 1:2)` or `data.table(y1 = 1:2, y2 = 3:4)`.
- `domain` (paradox::ParamSet)
  - Specifies domain of function. The paradox::ParamSet should describe all possible input parameters of the objective function. This includes their id, their types and the possible range.
- `codomain` (paradox::ParamSet)
  - Specifies codomain of function. Most importantly the tags of each output "Parameter" define whether it should be minimized or maximized. The default is to minimize each component.
- `id` (character(1)).
- `properties` (character()).
- `constants` (paradox::ParamSet)
  - Changeable constants or parameters that are not subject to tuning can be stored and accessed here.
- `check_values` (logical(1))
  - Should points before the evaluation and the results be checked for validity?

Method `eval_many()`: Evaluates input value(s) on the objective function. Calls the R function supplied by the user.

Usage:

```r
ObjectiveRFunMany$eval_many(xss)
```

Arguments:
xss (list())
   A list of lists that contains multiple x values, e.g. list(list(x1 = 1, x2 = 2), list(x1 = 3, x2 = 4)).

Returns: data.table::data.table() that contains one y-column for single-criteria functions and multiple y-columns for multi-criteria functions, e.g. data.table(y = 1:2) or data.table(y1 = 1:2, y2 = 3:4). It may also contain additional columns that will be stored in the archive if called through the OptimInstance. These extra columns are referred to as extras.

Method clone(): The objects of this class are cloneable with this method.

Usage:
ObjectiveRFunMany$clone(deep = FALSE)

Arguments:
   deep Whether to make a deep clone.

Examples

```
# define objective function
fun = function(xss) {
   res = lapply(xss, function(xs) -(xs[[1]] - 2)^2 - (xs[[2]] + 3)^2 + 10)
   data.table(y = as.numeric(res))
}

# set domain
domain = ps(
   x1 = p_dbl(-10, 10),
   x2 = p_dbl(-5, 5)
)

# set codomain
codomain = ps(y = p_dbl(tags = "maximize"))

# create Objective object
obfun = ObjectiveRFunMany$new(
   fun = fun,
   domain = domain,
   codomain = codomain,
   properties = "deterministic"
)
```
Usage

opt(.key, ...)  

opts(.keys, ...)

Arguments

.key (character(1))  
Key passed to the respective dictionary to retrieve the object.

... (named list())  
Named arguments passed to the constructor, to be set as parameters in the paradox::ParamSet, or to be set as public field. See mlr3misc::dictionary_sugar_get() for more details.

[keys (character())  
Keys passed to the respective dictionary to retrieve multiple objects.

Value

• Optimizer for opt().
• list of Optimizer for opts().

Examples

opt("random_search", batch_size = 10)

OptimInstance

Optimization Instance with budget and archive

Description

Abstract base class.

Technical details

The Optimizer writes the final result to the .result field by using the $assign_result() method. .result stores a data.table::data.table consisting of x values in the search space, (transformed) x values in the domain space and y values in the codomain space of the Objective. The user can access the results with active bindings (see below).

Public fields

objective (Objective).

search_space (paradox::ParamSet).

terminator (Terminator).

archive (Archive).
OptimInstance

progressor (progressor())
Store progressor function.
objective_multiplicator (integer()).
callbacks (List of CallbackOptimizations).

Active bindings
result (data.table::data.table)
Get result
result_x_search_space (data.table::data.table)
x part of the result in the search space.
result_x_domain (list())
(transformed) x part of the result in the domain space of the objective.
result_y (numeric())
Optimal outcome.
is_terminated (logical(1)).

Methods
Public methods:
• OptimInstance$new()
• OptimInstance$format()
• OptimInstance$print()
• OptimInstance$eval_batch()
• OptimInstance$assign_result()
• OptimInstance$objective_function()
• OptimInstance$clear()
• OptimInstance$clone()

Method new(): Creates a new instance of this R6 class.
Usage:
OptimInstance$new(
  objective,
  search_space = NULL,
  terminator,
  keep_evals = "all",
  check_values = TRUE,
  callbacks = list()
)
Arguments:
objective (Objective).
search_space (paradox::ParamSet)
  Specifies the search space for the Optimizer. The paradox::ParamSet describes either a subset of the domain of the Objective or it describes a set of parameters together with a trafo function that transforms values from the search space to values of the domain. Depending on the context, this value defaults to the domain of the objective.
OptimInstance

terminator (Terminator),
keep_evals (character(1))
    Keep all or only best evaluations in archive?
check_values (logical(1))
    Should x-values that are added to the archive be checked for validity? Search space that is
    logged into archive.
callbacks (list of mlr3misc::Callback)
    List of callbacks.

Method format(): Helper for print outputs.
Usage:
OptimInstance$format()

Method print(): Printer.
Usage:
OptimInstance$print(...)
Arguments:
... (ignored).

Method eval_batch(): Evaluates all input values in xdt by calling the Objective. Applies
possible transformations to the input values and writes the results to the Archive.
Before each batch-evaluation, the Terminator is checked, and if it is positive, an exception of class
terminated_error is raised. This function should be internally called by the Optimizer.
Usage:
OptimInstance$eval_batch(xdt)
Arguments:
xdt (data.table::data.table())
    x values as data.table() with one point per row. Contains the value in the search space
    of the OptimInstance object. Can contain additional columns for extra information.

Method assign_result(): The Optimizer object writes the best found point and estimated
performance value here. For internal use.
Usage:
OptimInstance$assign_result(xdt, y)
Arguments:
xdt (data.table::data.table())
    x values as data.table::data.table() with one row. Contains the value in the search
    space of the OptimInstance object. Can contain additional columns for extra information.
y (numeric(1))
    Optimal outcome.

Method objective_function(): Evaluates (untransformed) points of only numeric values.
Returns a numeric scalar for single-crit or a numeric vector for multi-crit. The return value(s) are
negated if the measure is maximized. Internally, $eval_batch() is called with a single row. This
function serves as a objective function for optimizers of numeric spaces - which should always be
minimized.
OptimInstanceMultiCrit

Usage:
OptimInstance$objective_function(x)

Arguments:

x (numeric())
  Untransformed points.

Returns: Objective value as numeric(1), negated for maximization problems.

Method clear(): Reset terminator and clear all evaluation results from archive and results.

Usage:
OptimInstance$clear()

Method clone(): The objects of this class are cloneable with this method.

Usage:
OptimInstance$clone(deep = FALSE)

Arguments:

deep  Whether to make a deep clone.

OptimInstanceMultiCrit

Optimization Instance with budget and archive

Description

Wraps a multi-criteria Objective function with extra services for convenient evaluation. Inherits from OptimInstance.

• Automatic storing of results in an Archive after evaluation.
• Automatic checking for termination. Evaluations of design points are performed in batches. Before a batch is evaluated, the Terminator is queried for the remaining budget. If the available budget is exhausted, an exception is raised, and no further evaluations can be performed from this point on.

Super class

bbotk::OptimInstance -> OptimInstanceMultiCrit

Active bindings

result_x_domain (list())
  (transformed) x part of the result in the domain space of the objective.

result_y (numeric(1))
  Optimal outcome.
Methods

Public methods:

• OptimInstanceMultiCrit$new()
• OptimInstanceMultiCrit$assign_result()
• OptimInstanceMultiCrit$clone()

Method new(): Creates a new instance of this R6 class.

Usage:
OptimInstanceMultiCrit$new(
  objective,
  search_space = NULL,
  terminator,
  keep_evals = "all",
  check_values = TRUE,
  callbacks = list()
)

Arguments:

objective (Objective).

search_space (paradox::ParamSet)
  Specifies the search space for the Optimizer. The paradox::ParamSet describes either a
  subset of the domain of the Objective or it describes a set of parameters together with
  a trafo function that transforms values from the search space to values of the domain.
  Depending on the context, this value defaults to the domain of the objective.

terminator (Terminator)
  Multi-criteria terminator.

keep_evals (character(1))
  Keep all or only best evaluations in archive?

check_values (logical(1))
  Should x-values that are added to the archive be checked for validity? Search space that is
  logged into archive.

callbacks (list of mlr3misc::Callback)
  List of callbacks.

Method assign_result(): The Optimizer object writes the best found points and estimated
performance values here (probably the Pareto set / front). For internal use.

Usage:
OptimInstanceMultiCrit$assign_result(xdt, ydt)

Arguments:

xdt (data.table::data.table())
  Set of untransformed points / points from the search space. One point per row, e.g. data.table(x1
  = c(1, 3), x2 = c(2, 4)). Column names have to match ids of the search_space. However, xdt can contain additional columns.

ydt (numeric(1))
  Optimal outcomes, e.g. the Pareto front.
Method clone(): The objects of this class are cloneable with this method.

Usage:
OptimInstanceMultiCrit$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

OptimInstanceSingleCrit
Optimization Instance with budget and archive

Description
Wraps a single-criteria Objective function with extra services for convenient evaluation. Inherits from OptimInstance.

• Automatic storing of results in an Archive after evaluation.
• Automatic checking for termination. Evaluations of design points are performed in batches. Before a batch is evaluated, the Terminator is queried for the remaining budget. If the available budget is exhausted, an exception is raised, and no further evaluations can be performed from this point on.

Super class
bbotk::OptimInstance -> OptimInstanceSingleCrit

Methods
Public methods:
• OptimInstanceSingleCrit$new()
• OptimInstanceSingleCrit$assign_result()
• OptimInstanceSingleCrit$clone()

Method new(): Creates a new instance of this R6 class.

Usage:
OptimInstanceSingleCrit$new(
  objective,
  search_space = NULL,
  terminator,
  keep_evals = "all",
  check_values = TRUE,
  callbacks = list()
)

Arguments:
objective (Objective).
search_space \textbf{(paradox::ParamSet)}

Specifies the search space for the Optimizer. The paradox::ParamSet describes either a subset of the domain of the Objective or it describes a set of parameters together with a trafo function that transforms values from the search space to values of the domain. Depending on the context, this value defaults to the domain of the objective.

terminator (Terminator).

keep_evals (character(1))

Keep all or only best evaluations in archive?

check_values (logical(1))

Should x-values that are added to the archive be checked for validity? Search space that is logged into archive.

callbacks (list of mlr3misc::Callback)

List of callbacks.

\textbf{Method} assign_result(): The Optimizer object writes the best found point and estimated performance value here. For internal use.

\textit{Usage:}

OptimInstanceSingleCrit$assign_result(xdt, y)

\textit{Arguments:}

\textit{xdt (data.table::data.table())}

Set of untransformed points / points from the \textit{search space}. One point per row, e.g. data.table(x1 = c(1, 3), x2 = c(2, 4)). Column names have to match ids of the search_space. However, xdt can contain additional columns.

\textit{y (numeric(1))}

Optimal outcome.

\textbf{Method} clone(): The objects of this class are cloneable with this method.

\textit{Usage:}

OptimInstanceSingleCrit$clone(deep = FALSE)

\textit{Arguments:}

deep Whether to make a deep clone.

---

\textbf{Optimizer} \hspace{1cm} Optimizer

\textbf{Description}

Abstract Optimizer class that implements the base functionality each Optimizer subclass must provide. A Optimizer object describes the optimization strategy. A Optimizer object must write its result to the $assign_result() method of the OptimInstance at the end in order to store the best point and its estimated performance vector.
Optimizer

Progress Bars

$optimize() supports progress bars via the package progressr combined with a Terminator. Simply wrap the function in progressr::with_progress() to enable them. We recommend to use package progressr as backend; enable with progressr::handlers("progress").

Public fields

id (character(1))
Identifier of the object. Used in tables, plot and text output.

Active bindings

param_set paradox::ParamSet
Set of control parameters.

label (character(1))
Label for this object. Can be used in tables, plot and text output instead of the ID.

man (character(1))
String in the format [pkg]:[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().

param_classes (character())
Supported parameter classes that the optimizer can optimize. Subclasses of paradox::Param.

properties (character())
Set of properties of the optimizer. Must be a subset of bbotk_reflections$optimizer_properties.

packages (character())
Set of required packages. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via requireNamespace()

Methods

Public methods:

- Optimizer$new()
- Optimizer$format()
- Optimizer$print()
- Optimizer$help()
- Optimizer$optimize()
- Optimizer$clone()

Method new(): Creates a new instance of this R6 class.

Usage:
Optimizer$new(
  id = "optimizer",
  param_set,
  param_classes,
  properties,
  packages = character(),
  label = NA_character_,
)
man = NA_character_
)

Arguments:
id (character(1))
   Identifier for the new instance.
param_set (paradox::ParamSet)
   Set of control parameters.
param_classes (character())
   Supported parameter classes that the optimizer can optimize. Subclasses of paradox::Param.
properties (character())
   Set of properties of the optimizer. Must be a subset of bbotk_reflections$optimizer_properties.
packages (character())
   Set of required packages. A warning is signaled by the constructor if at least one of the packages is not installed, but loaded (not attached) later on-demand via requireNamespace().
label (character(1))
   Label for this object. Can be used in tables, plot and text output instead of the ID.
man (character(1))
   String in the format [pkg]:[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().

Method format(): Helper for print outputs.
Usage:
Optimizer$format()

Method print(): Print method.
Usage:
Optimizer$print()
Returns: (character()).

Method help(): Opens the corresponding help page referenced by field $man.
Usage:
Optimizer$help()

Method optimize(): Performs the optimization and writes optimization result into OptimInstance. The optimization result is returned but the complete optimization path is stored in Archive of OptimInstance.
Usage:
Optimizer$optimize(inst)
Arguments:
inst (OptimInstance).
Returns: data.table::data.table.

Method clone(): The objects of this class are cloneable with this method.
Usage:
Optimizer$clone(deep = FALSE)
Arguments:
deep Whether to make a deep clone.
Description

Wraps progressr::progressor() function and stores current progress.

Public fields

progressor (progressr::progressor()).
max_steps (integer(1)).
current_steps (integer(1)).
unit (character(1)).

Methods

Public methods:
• Progressor$new()
• Progressor$update()
• Progressor$clone()

Method new(): Creates a new instance of this R6 class.

Usage:
Progressor$new(progressor, unit)

Arguments:
progressor (progressr::progressor())
Progressor function.
unit (character(1))
Unit of progress.

Method update(): Updates progressr::progressor() with current steps.

Usage:
Progressor$update(terminator, archive)

Arguments:
terminator (Terminator).
archive (Archive).

Method clone(): The objects of this class are cloneable with this method.

Usage:
Progressor$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.
Shrink a ParamSet towards a point.

Description

Shrinks a paradox::ParamSet towards a point. Boundaries of numeric values are shrinked to an interval around the point of half of the previous length, while for discrete variables, a random (currently not chosen) level is dropped.

Note that for paradox::ParamLglS the value to be shrinked around is set as the default value instead of dropping a level. Also, a tag shrinked is added.

Note that the returned paradox::ParamSet has lost all its original defaults, as they may have become infeasible.

If the paradox::ParamSet has a trafo, x is expected to contain the transformed values.

Usage

shrink_ps(param_set, x, check.feasible = FALSE)

Arguments

- `param_set` (paradox::ParamSet) The paradox::ParamSet to be shrinked.
- `x` (data.table::data.table) data.table::data.table with one row containing the point to shrink around.
- `check.feasible` (logical(1)) Should feasibility of the parameters be checked? If feasibility is not checked, and invalid values are present, no shrinking will be done. Must be turned off in the case of the paradox::ParamSet having a trafo. Default is FALSE.

Value

paradox::ParamSet

Examples

library(paradox)
library(data.table)
param_set = ParamSet$new(list(
  ParamDbl$new("x1", lower = 0, upper = 10),
  ParamInt$new("x2", lower = -10, upper = 10),
  ParamFct$new("x3", levels = c("a", "b", "c")),
  ParamLgl$new("x4")
))
x = data.table(x1 = 5, x2 = 0, x3 = "b", x4 = FALSE)
shrink_ps(param_set, x = x)
Description

Abstract Terminator class that implements the base functionality each terminator must provide. A terminator is an object that determines when to stop the optimization.

Termination of optimization works as follows:

• Evaluations in an instance are performed in batches.
• Before each batch evaluation, the Terminator is checked, and if it is positive, we stop.
• The optimization algorithm itself might decide not to produce any more points, or even might decide to do a smaller batch in its last evaluation.

Therefore the following note seems in order: While it is definitely possible to execute a fine-grained control for termination, and for many optimization algorithms we can specify exactly when to stop, it might happen that too few or even too many evaluations are performed, especially if multiple points are evaluated in a single batch (c.f. batch size parameter of many optimization algorithms). So it is advised to check the size of the returned archive, in particular if you are benchmarking multiple optimization algorithms.

Technical details

Terminator subclasses can overwrite .status() to support progress bars via the package progressr. The method must return the maximum number of steps (max_steps) and the currently achieved number of steps (current_steps) as a named integer vector.

Public fields

id (character(1))
   Identifier of the object. Used in tables, plot and text output.

Active bindings

param_set paradox::ParamSet
   Set of control parameters.

label (character(1))
   Label for this object. Can be used in tables, plot and text output instead of the ID.

man (character(1))
   String in the format [pkg]::[topic] pointing to a manual page for this object. The referenced help package can be opened via method $help().

properties (character())
   Set of properties of the terminator. Must be a subset of bbotk_reflections$terminator_properties.

unit (character())
   Unit of steps.
Methods

Public methods:

- `Terminator$new()`
- `Terminator$format()`
- `Terminator$print()`
- `Terminator$status()`
- `Terminator$remaining_time()`
- `Terminator$clone()`

Method `new()`: Creates a new instance of this R6 class.

Usage:

```r
Terminator$new(id, param_set = ps(), properties = character(), unit = "percent", label = NA_character_, man = NA_character_)
```

Arguments:

- `id` (character(1))  
  Identifier for the new instance.
- `param_set` (paradox::ParamSet)  
  Set of control parameters.
- `properties` (character())  
  Set of properties of the terminator. Must be a subset of `bbotk_reflections$terminator_properties`.
- `unit` (character())  
  Unit of steps.
- `label` (character(1))  
  Label for this object. Can be used in tables, plot and text output instead of the ID.
- `man` (character(1))  
  String in the format `[pkg]::[topic]` pointing to a manual page for this object. The referenced help package can be opened via method `$help()`.

Method `format()`: Helper for print outputs.

Usage:

```r
Terminator$format(with_params = FALSE)
```

Arguments:

- `with_params` (logical(1))  
  Add parameter values to format string.

Method `print()`: Printer.

Usage:

```r
Terminator$print(...)
```
Arguments:
... (ignored).

Method status(): Returns how many progression steps are made (current_steps) and the amount steps needed for termination (max_steps).

Usage:
Terminator$status(archive)

Arguments:
archive (Archive).

Returns: named integer(2).

Method remaining_time(): Returns remaining runtime in seconds. If the terminator is not time-based, the remaining runtime is Inf.

Usage:
Terminator$remaining_time(archive)

Arguments:
archive (Archive).

Returns: integer(1).

Method clone(): The objects of this class are cloneable with this method.

Usage:
Terminator$clone(deep = FALSE)

Arguments:
deep Whether to make a deep clone.

See Also

Description
This function complements mlr_terminators with functions in the spirit of mlr_sugar from mlr3.

Usage
trm(.key, ...)

trms(.keys, ...)

Syntactic Sugar Terminator Construction
Arguments

.key (character(1))
Key passed to the respective dictionary to retrieve the object.

...(named list())
Named arguments passed to the constructor, to be set as parameters in the param::ParamSet, or to be set as public field. See mlr3misc::dictionary_sugar_get() for more details.

.keys (character())
Keys passed to the respective dictionary to retrieve multiple objects.

Value

- Terminator for trm().
- list of Terminator for trms().

Examples

trm("evals", n_evals = 10)
Index

* Dictionary
  * mlr_optimizers, 18
* Optimizer
  * mlr_optimizers, 18
* Terminator
  * mlr_terminators, 34
  * mlr_terminators_clock_time, 35
  * mlr_terminators_combo, 37
  * mlr_terminators_evals, 39
  * mlr_terminators_none, 40
  * mlr_terminators_perf_reached, 42
  * mlr_terminators_run_time, 43
  * mlr_terminators_stagnation, 45
  * mlr_terminators_stagnation_batch, 46
  * Terminator, 68
* datasets
  * mlr_optimizers, 18
  * mlr_terminators, 34
  * adagio::pureCMAES(), 18, 19
  * Archive, 4, 7, 8, 14, 28, 36, 38, 40–42, 44, 45, 47, 57, 60, 62, 65, 66, 70
  * ArchiveBest, 7, 7
  * bb_optimize, 9
  * bbotk (bbotk-package), 3
  * bbotk-package, 3
  * bbotk.backup, 8
  * bbotk::Archive, 7
  * bbotk::Objective, 50, 52, 54
  * bbotk::OptimInstance, 60, 62
  * bbotk::Optimizer, 19, 21, 23, 25, 27, 29, 31, 33
  * bbotk::Terminator, 36, 37, 39, 41–43, 45, 46
  * bbotk_reflections$optimizer_properties, 64, 65
  * bbotk_reflections$terminator_properties, 68, 69
  * branin, 11
  * branin_wu (branin), 11
  * callback_optimization, 13
  * callback_optimization(), 12, 16
  * CallbackOptimization, 8, 12, 12, 13, 58
  * clbk(), 12
  * Codomain, 4, 14
  * ContextOptimization, 13, 14, 16, 16
  * data.table::data.table, 4, 16, 21, 38, 57, 58, 65, 67
  * data.table::data.table(), 4–6, 8, 18, 35, 50, 54–56, 61, 63
  * dictionary, 12, 19, 21, 22, 24, 26, 29, 33, 35, 37, 39, 41–43, 45, 46, 57, 71
  * GenSA::GenSA(), 24
  * irace::defaultScenario(), 28
  * irace::irace(), 28
  * is_dominated, 17
  * mlr3misc::Callback, 12, 16, 59, 61, 63
  * mlr3misc::Context, 16
  * mlr3misc::Dictionary, 18, 34, 35
  * mlr3misc::dictionary_sugar_get(), 57, 71
  * mlr_callsbacks, 12
  * mlr_optimizers, 9, 18, 19, 21, 22, 24, 26, 29, 33, 36
  * mlr_optimizers_cmaes, 18
  * mlr_optimizers_design_points, 20
  * mlr_optimizers_focus_search, 22
  * mlr_optimizers_gensa, 24
  * mlr_optimizers_grid_search, 26
  * mlr_optimizers_irace, 28
  * mlr_optimizers_nloptr, 31
  * mlr_optimizers_random_search, 33
  * mlr_terminators, 34, 35–47, 70
  * mlr_terminators_clock_time, 35, 35, 38, 40, 41, 43, 44, 46, 47, 70
mlr_terminators_combo, 35, 36, 37, 40, 41, 43, 44, 46, 47, 70
mlr_terminators_evals, 35, 36, 38, 40, 43, 44, 46, 47, 70
mlr_terminators_none, 35, 36, 38, 40, 43, 44, 46, 47, 70
mlr_terminators_perf_reached, 35, 36, 38, 40, 41, 42, 44, 46, 47, 70
mlr_terminators_run_time, 35, 36, 38, 40, 41, 43, 44, 46, 47, 70
mlr_terminators_stagnation, 35, 36, 38, 40, 41, 43, 44, 45, 47, 70
mlr_terminators_stagnation_batch, 35, 36, 38, 40, 41, 43, 44, 45, 46, 47, 70

nloptr::nloptr(), 31
nloptr::nloptr.print.options(), 31

Objective, 4, 5, 7, 9, 10, 14, 28, 48, 57–63
ObjectiveRFun, 50
ObjectiveRFunDt, 52
ObjectiveRFunMany, 54
opt, 56
opt(), 18, 19, 21, 22, 24, 26, 29, 33
OptimInstance, 16, 17, 49, 50, 56, 57, 59, 60, 62, 63, 65
OptimInstanceMultiCrit, 10, 60
OptimInstanceSingleCrit, 10, 62
Optimizer, 4, 5, 7, 9, 10, 16–19, 21, 22, 24, 26, 29, 33, 44, 45–57, 59, 61, 63, 65
OptimizerCmaes (mlr_optimizers_cmaes), 18
OptimizerDesignPoints
 (mlr_optimizers_design_points), 20
OptimizerFocusSearch
 (mlr_optimizers_focus_search), 22
OptimizerGenSA (mlr_optimizers_gensa), 24
OptimizerGridSearch, 40
OptimizerGridSearch
 (mlr_optimizers_grid_search), 26
OptimizerIrace (mlr_optimizers_irace), 28
OptimizerNloptr
 (mlr_optimizers_nloptr), 31

OptimizerRandomSearch
 (mlr_optimizers_random_search), 33
opts (opt), 56
opts(), 18

paradox::generate_design_grid(), 26
paradox::Param, 64, 65
paradox::ParamLgl, 67
paradox::ParamSet, 4, 5, 7, 10, 14, 15, 48, 49, 51, 53, 55, 57, 58, 61, 63–65,
67–69, 71
Param, 14, 15
POSIXct, 4
Progressor, 66

R6, 5, 7, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 36, 37, 39, 41, 42, 44, 45, 47, 48, 51,
53, 55, 58, 61, 62, 64, 66, 69
R6::R6Class, 18, 35
requireNamespace(), 64, 65

shrink_ps, 22, 67
Sys.time(), 35

TerminatorClockTime, 7
TerminatorClockTime
 (mlr_terminators_clock_time), 35
TerminatorCombo, 10
TerminatorCombo
 (mlr_terminators_combo), 37
TerminatorEvals, 7, 28
TerminatorEvals
 (mlr_terminators_evals), 39
TerminatorNone, 7, 10
TerminalNone (mlr_terminators_none), 40
TerminatorPerfReached
 (mlr_terminators_perf_reached), 42
TerminatorRunTime
 (mlr_terminators_run_time), 43
TerminatorStagnation
 (mlr_terminators_stagnation), 45
TerminatorStagnationBatch (mlr_terminators_stagnation_batch), 46

trm, 70
trm(), 34, 35, 37, 39, 41–43, 45, 46
trms (trm), 70
trms(), 34, 35